<알고리즘 Mix Tape 03>

7장 동적 프로그래밍 (170515 to 170522)

알고리즘 공부를 잠깐 뒤로 하고 오락실 게임을 살펴보자. 예를 들어 동키콩(고릴라가 마리오를 향해 이상한거 굴러대는 거로 알 것이다.) 같이 최고 점수 내는 게임인 경우에는 실제로 컨티뉴(동전 넣으면 계속 이어하는 원리) 기능도 없었으며 중간 저장 기능이 없어서 맨 처음부터 다시 사다리를 올라타면서 개고생하는 게임과 슈퍼마리오나 소닉처럼 스테이지 별로 중간 저장점이 있으면서 이어하는 기능이 추가되는 원리를 생각해보면 중간 저장점이 있는 기능이 바로 동적 프로그래밍의 간단한 원리라고 생각할 수 있겠다. 실제로 알고리즘에서도 재귀 호출로 떡칠되어 있으면 중복이 워낙 많이 되어 어떤 변수에 저장을 해둠으로서 중복을 줄여 나가는 원리가 바로 동적 프로그래밍의 원리로 생각을 할 수 있겠다. 이번 요약정리에서는 동적 프로그램의 원리에 대해서 살펴보고, 실제로 사용되는 예를 들으면서 정리할 것이다.

1. 재귀 호출의 추억

우리가 알고리즘 초반에 공부하였을 때 잠깐 재귀 호출을 이용해서 메모리를 그리면서 공부를 해본 적이 있었다. 이 뿐만 아니라 병합 정렬, 퀵 정렬처럼 재귀 호출이 필요한 정렬 알고리즘뿐만 아니라 이진 탐색 트리, 연결 리스트, 앞으로 배울 그래프 등등에도 재귀 호출에 대해 다룬 메소드들이 많이 등장하고 있는 추세이다. 그래서 재귀적 문제에 대해서 숙지를 하고 넘어가야 겠다.

1-1) 재귀적 문제를 구현하기 위해서

재귀적 문제에 대해 생각을 하기 위해서 아래 3가지를 고려를 해야겠다.
0> 공통적으로 쓰이는 함수를 생각한다.
1> 이전 단계의 답을 이용해서 해결을 한다.
2> 현재 단계에서 작업(계산)을 구현한다.
3> 종료 조건을 구현한다.
이처럼 3단계로 나뉘어서 해결을 한다면 재귀 호출에 대해서 생각을 해볼 수 있겠다. 이를 실생활에서 생각을 해보면 도움이 되겠다.

우리가 예를 들어 지하철을 타고 수원역에서 안양역까지 이동한다고 가정을 하자.

세	병	세	수	화	성	의	당	군	금	명	안	석
마	점	류	원	서	대	왕	정	포	정	학	양	수

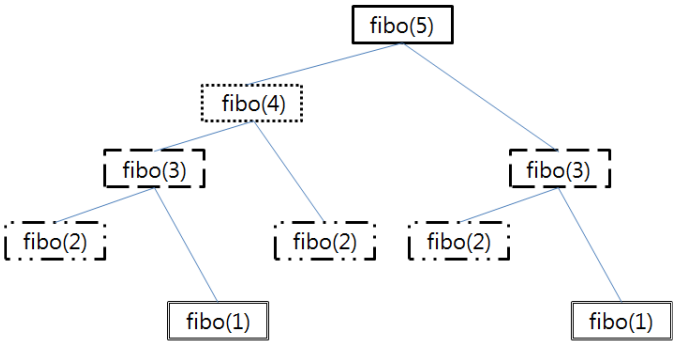
만일 수원역에서 출발을 하게 되면 천안 방면으로 가는 전철이 있을 것이고, 구로 방면(쉽게 생각하기 위해 구로행만 있다고 가정을 하자.)으로 가는 전철이 있을 것이다.
-> 공통적으로 쓰이는 함수를 만드는 과정
그러면 안양역으로 가기 위해서는 천안역이든 병점역이든 간에 안양역 이전에는 구로행 열차를 탑승해야 안양역까지 접근이 가능하다. 그러면 우리가 공통적으로 쓰이는 함수를 여기서는 [구로행_다음역_운행]으로 잡고 시작을 하겠다.
-> 이전 단계의 답을 이용해서 해결하기
여기서 수원역 다음에 하차하면 화서역이 되고, 또 화서역에서 탑승을 하면 성균관대에서 내리게 된다. [구로행_운행] 함수는 각 한 역으로 이동하는 과정이 이전 과정으로 생각을 하면 수원역에서 안양역까지 이동을 하는데 명학역부터 화서역까지 모두 봐야 도착을 하게 된다. 이를 목적지에 도착하기 이전인 이전 단계의 답으로 잡을 수가 있다.
구로행_다음역_운행(안양역){ // 이 과정이 안양역 이전에 모든 역을 구경하는 과정이다.
  구로행_다음역_운행(명학역) // 이전 작업이 명학역 이전에 모든 역을 구경하는 과정으로 볼 수 있겠다.
}
-> 현재 단계에서 작업 구현하는 과정
그러면 우리가 현재 단계에 대해 구현을 하는 것이 [역_도착]으로 생각을 해볼 수 있겠다. 그러면 다음과 같이 표현이 된다.
구로행_다음역_운행(안양역){
  역_도착(안양역)+구로행_다음역_운행(명학역);
// 이 다음역_운행 메소드가 쉽게 생각하면 안양역에 도착을 함과 동시에 명학역에서 계속 이동하는 방식으로 구현이 된다고 생각을 해볼 수 있겠다.
}
-> 종료 조건을 구현하자.
재귀 함수는 애석하게도 모든 과정이 아니지만 대부분 역으로 생각을 해야 된다. 우리가 출발한 역은 수원역이니 종료되는 역을 수원역으로 잡게 되면 [역_출발]로 생각을 해볼 수 있겠다.
구로행_다음역_운행(안양역){
  if(수원역) 역_출발(수원역);
  else 역_도착(안양역)+구로행_다음역_운행(명학역);
}
이처럼 실생활에서 쓰이는 재귀적인 구조에 대해 생각을 하면 알고리즘에서 실제로 쓰이는 구현에 대해서 사례를 공부하는데 있어서 큰 도움이 될 것이다.

1-2) 재귀 호출의 단점을 찾아서

우리가 피보나치 수에 대해 잠깐 고려를 해본 적이 있을 것이다. 피보나치는 황금비(1:1.618)랑 연관이 있는 수열이라 생각해볼 수 있겠다. 이는 아래처럼 계산이 된다.
f(1), f(2) : 1 / f(3) : f(2)+f(1)=2 / f(4) : f(3)+f(2)=3 / f(5) : f(4)+f(3)=5 / f(6) : f(5)+f(4)=8 …
f(n) : f(n-1)+f(n-2) (n≠1, n≠2)
그러면 재귀로 표현을 하면 아래처럼 작성이 된다.

```
public static int fibo(int n){
  System.out.printf(" fibo(%d) / ", n); // ?
  if(n==1 || n==2) return 1;
  else return fibo(n-1)+fibo(n-2);
}
```

예를 들어 f(5)를 넣어보도록 하자. 그러면 5가 나오기 이전에 ? 부분이 나오게 된다. 그 ? 부분을 분석해보도록 하자. ? 부분에는
fibo(5) / fibo(4) / fibo(3) / fibo(2) / fibo(1) / fibo(2) /
1번째 구분
fibo(3) / fibo(2) / fibo(1) /
2번째 구분
이렇게 결과가 나오게 된다. fibo(5)를 구하는데 어떻게 많은 결과가 나오게 된 것일까?



우리가 재귀 호출을 하게 되면 이처럼 중복 호출을 많이 부르게 되는 경우가 부지기수하게 많다. 그래서 같은 매개 변수에 대한 낭비(특히 2처럼)를 줄이기 위한 방법으로서는 어딘가에 저장을 하는 방법에 대해서 고려를 해야겠다. 이러한 방안이 바로 동적 프로그래밍이다.

2. 동적 프로그래밍

아까 재귀적 문제를 재귀호출로 계속 구현하게 되면 fibo(2)처럼 중복이 허다한 경우를 살펴볼 수 있다. 이처럼 어딘가에 저장을 하면서 중복을 줄이는 방안이 바로 동적 프로그래밍이라고 할 수 있겠다. 동적 프로그래밍은 어떤 원리로 돌아가는지에 대해 아까 피보나치 수열을 통해서 공부해보도록 하자.

2-1) 동적 프로그래밍으로 해결하기

```
static int[] arr=new int[100];
public static int fibo(int n){
   System.out.printf("fibo(%d)\n", n);
   if(arr[n]!=0) return arr[n]; // 1
   if(n==1 || n==2) arr[n]=1;
   else arr[n]=fibo(n-1)+fibo(n-2); // 2
   return arr[n];
}
```

1 : 동적 프로그래밍의 정석으로서는 시간 복잡도를 줄이는데 목적이 있지만 공간 복잡도에 대해서는 둘째 치라는 뜻으로 받아들일 수 있겠다. 여기서 n번째 배열의 요소가 존재한다는 것은 이미 계산을 해서 저장을 했다는 의미로 생각할 수 있겠다. 그래서 이미 계산이 완료된 부분에 대해서는 1번 아래 부로 넘어가지 않고 값을 반환하는 것으로 끝낸다는 점을 생각하면 쉽겠다.
2 : 이는 아까 피보나치 수열에 대해서 계산한 원리로 생각을 할 수 있겠다. 아까와 다른 점은 n 별로 각각 값을 계산해서 저장을 하는 원리로 생각을 해보면 되겠다. 하지만 이처럼 계산을 하는 것은 공간 복잡도에 대한 보장은 못 할지라도 시간 복잡도는 줄여준다는 의미로 공부를 하면 되겠다.
위 메소드 n에 5를 넣으면 아까와 달리
fibo(5) -> 가
fibo(4) -> 나
fibo(3) -> 다
fibo(2) -> 라
fibo(1) -> 마
fibo(2) -> 바
fibo(3) -> 사
이 나오게 된다. 이렇게 나오는 이유는 n의 값에 따라서 재귀를 하게 되면 아래처럼 저장이 되는 현상을 살펴볼 수 있기 때문이다.

	fibo(1)	fibo(2)	fibo(3)	fibo(4)	fibo(5)
저장값	1	1	2	3	5
공간	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]

그럼 아까 fibo(1)이나 fibo(2)처럼 값을 반환하는 데이터에 대해서는 어쩔 수 없이 작동을 해야 되지만, fibo(3)부터 필요로 한 값을 반환하는데 있어서 fibo(1)이나 fibo(2)를 반환해서 초반부에만 값을 불러와서 계산을 해주고 결국엔 배열의 값만 불러오게 되니깐 간단하게 해결되는 점을 살펴볼 수 있겠다. 이처럼 동적 프로그래밍을 이용해서 우리가 수업 시간에 많이 다루게 될 3가지 문제(경로 구하기, 조약돌 두기, 행렬 곱셈)를 사례로 동적 프로그래밍과 재귀 호출의 차이를 느껴보도록 하자.

3. 경로 구하기 문제

이제 실질적으로 재귀 함수와 동적 프로그래밍을 이용해서 행렬 경로 문제에 대해 접근을 해보도록 하자. 강의 노트에는 문제가 이해가 가지 않기 때문에 거북이를 비유해서 문제를 쉽게 이해해보는 연습을 하면 도움이 될 듯하다. 아래의 문제를 살펴보도록 하자.

<행렬 경로 구하기 문제>
거북이가 행의 크기랑 열의 크기가 같은 행렬 위에 있다고 가정하자. 각 행렬의 칸에는 거북이가 좋아하는 사료들의 개수가 써져 있고, (0, 0)번째 칸부터 시작해서 (n, n)칸으로 내려오는 조건을 대신에 방문한 칸에 있는 사료들을 먹으면서 **<u>많이 먹게 되면 병 걸리기 때문에 최소만큼 먹은 경로를 찾는 것</u>**이 관건이다.

4	2	3
5	1	9
7	3	8

대신에 제약 조건이 따른다.
1> 거북이는 각 칸에 있는 사료들의 개수를 남김없이 모두 먹어야 이동할 수 있다.
2> 거북이는 오른쪽, 아래쪽 두 방향으로만 이동이 가능하다. 왼쪽, 위, 대각선 방향으로 이동이 불가능하다.
3> 거북이가 이동을 하는 행위에 대해서만 집중을 하도록 하자. 중간에 쉬거나 볼일 보는 일 등 거북이의 생리적 문제에 대해서는 생략한다. 또한 거북이는 행렬 내부에서만 왔다리 갔다리 할 수 있다는 점을 제약 조건으로 추가하겠다.

강의 노트에 행렬 경로 문제와 매우 유사해서 문제를 푸는데 큰 변함은 없다. 이제 이러한 문제를 통해 어떻게 접근을 할지에 대해 생각을 해보도록 하자.

-> Next Page ->

3-1) 재귀적인 방법의 풀이

우선 거북이는 2차원 배열 내부에서만 왔다 갔다 할 수 있기 때문에 위 문제를 기준으로 거북이의 활동 범위는 (0, 0) 부터 (2, 2)까지 임을 숙지할 수 있을 것이다. 그래서 이 범위 이내에서 왔다리 갔다리를 하는 케이스는 4가지로서 나뉠 수 있다.

A) 거북이가 (0, 0)칸에 있는 경우

그러면 거북이는 어쩔 수 없이 아래쪽이나 오른쪽 두 방향 중 하나를 골라서 이동하는 방법이 있다. 그래서 여기서는 거북이가 (0, 0) 칸에 있는 사료의 개수만큼 먹고 출발을 하는 방법 이외엔 없다. 그러면 거북이가 출발 전에 먹은 사료의 개수는 0이 된다.

```
if(x==0 && y==0) eating_before=0;
```

B) 거북이가 (0, n) (n은 1이상의 자연수) 칸에 있는 경우

거북이는 이동이 가능한데 (0, n-1) 칸부터 이동을 하였을 경우 이외에 경로가 나오지 않았기 때문에 0번째 행에서 해결하는 방법밖에 없다. 그러면 거북이가 여태껏 먹은 사료의 개수는 0번째 행에서 총 먹었던 사료의 개수로 짐작이 된다.

```
else if(x==0 && y>0)
  eating_before=min_eating(x, y-1);
```

C) 거북이가 (n, 0) (n은 1 이상의 자연수) 칸에 있는 경우

거북이는 이동이 가능한데 (n-1, 0) 칸부터 이동을 하였을 경우 이외에 경로가 나오지 않았기 때문에 0번째 열에서 해결하는 방법밖엔 없다. 그래서 거북이가 여태껏 먹은 사료의 개수는 0번째 열에서 총 먹었던 사료의 개수로 짐작이 된다.

```
else if(x>0 && y==0)
  eating_before=min_eating(x-1, y);
```

D) 거북이가 임의의 칸에 있는 대신 위 칸도 있고 왼쪽 칸이 있는 경우

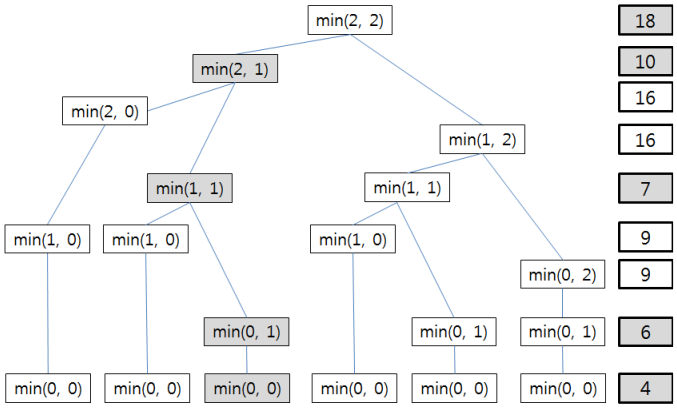
거북이는 (m, n) 칸에 이르는 방법이 왼쪽에서 오는 방법과 위에서 오는 방법 2가지로 나뉘게 된다. 여기서 최소한 먹은 양을 구하는 방법은 위에서 적게 먹고 왔느냐 아니면 왼쪽에서 적게 먹고 왔느냐를 따져봐야 할 때이다.

```
else
  eating_before=Math.min(min_eating(x-1, y),
min_eating(x, y-1));
```

그러면 이전에 먹은 양에 대해서 쉽게 따질 수 있게 되어 재귀적인 틀이 나오게 된다. 그럼 아래의 문장을 살펴보도록 하자.

```
static int[][] arr={{4, 2, 3}, {5, 1, 9}, {7, 3, 8}};
static int min_eating(int x, int y){
  int before_eating=0;
  if(x==0 && y==0) before_eating=0;
  else if(x==0 && y>0)
    before_eating=min_eating(x, y-1);
  else if(x>0 && y==0)
    before_eating=min_eating(x-1, y);
  else before_eating=Math.min(min_eating(x, y-1),
min_eating(x-1, y));
  return before_eating+arr[x][y];
}
```

그런데 재귀적으로 호출을 하게 된다면 많은 중복이 이루어지게 되기 때문에 생각보다 어렵게 된다. 거북이의 경로를 재귀적으로 생각을 해본다면 아래처럼 그려진다.



여기 오른쪽에 적혀있는 숫자는 각 칸 별로 거북이가 그 칸으로 이동을 하면서 사료를 최소로 먹을 수 있는 개수를 적었으며 우리가 문제를 해결하는데 거북이가 최소의 개수로 왔다리 갔다리 한 경우는 회색의 경우를 지났을 경우를 생각해볼 수 있겠다. 그럼 여기서는 중복의 수가 0, 0을 보면 많은 느낌이 들 것이다. 이를 줄이기 위해서 배열 하나를 만들고 접근을 해보도록 하자.

3-2) 동적 프로그래밍으로 풀이하기

```
static int[][] arr={{4, 2, 3}, {5, 1, 9}, {7, 3, 8}};
static int[][] sum=new int[3][3];
static int min_eating(int x, int y){
  if(sum[x][y]!=0) return sum[x][y]; // *
  int before_eating=0;
  if(x==0 && y==0) before_eating=0;
  else if(x==0 && y>0)
    before_eating=min_eating(x, y-1);
  else if(x>0 && y==0)
    before_eating=min_eating(x-1, y);
  else before_eating=Math.min(min_eating(x, y-1),
min_eating(x-1, y));
  sum[x][y]=before_eating+arr[x][y];
  return sum[x][y];
}
```

이처럼 동적 프로그래밍을 적용하게 되면 거북이가 경로를 이동하면서 최소로 먹을 수 있는 양을 sum 배열에 저장을 함으로서 중복을 줄여서 나오는 결과를 얻을 수 있게 된다. 실질적으로 중복을 줄여주는 이유가 바로 *에 있는 문장을 통해서 참고를 해보면 도움이 되겠다. 또한 sum 배열을 통해서 각 배열의 요소는 0, 0에서 각 칸으로 거북이가 이동을 하게 되면서 거북이가 최소로 먹은 사료의 총 개수를 표현해준다는 뜻도 알고 넘어가면 좋겠다.

3-3) 거북이가 이동한 경로를 표기해보자

```
static boolean min_eating_line(int x, int y){
  if(x==0 && y==0){
    System.out.println("<0, 0>");
    return true;
  }
  int before_eating=sum[x][y]-arr[x][y];
  if((x>0 && sum[x-1][y]==before_eating &&
min_eating_line(x-1, y))||(y>0 &&
sum[x][y-1]==before_eating && min_eating_line(x, y-1))){
    System.out.printf("<%d, %d>\n", x, y);
    return true;
  }
  return false;
}
```

이 코드에 대해서 자세하게 이해하기 위하여 과정 별로 소개하겠다.
1) min_eating_line(2, 2)를 실행하게 되면, before_eating의 값은 18-8이 되어 10이 된다. 그럼 여기서 (2, 1)번째 칸의 최소의 수가 같게 되니깐 min_eating_line(2, 1)이 다시 실행된다.
2) 그럼 before_eating의 값은 10-3=7이 되어 (1, 1)번째 칸의 최소의 수가 같게 되니깐 min_eating_line(1, 1)이 실행된다.
3) before_eating의 값은 6이 되니깐 (0, 1)번째 칸의 최소의 수가 같게 되니깐 min_eating(0, 1)이 실행된다.
4) (0, 1) 번째 칸에서 또 계산을 하면 (0, 0)번째 칸을 실행을 하게 되어 결국에는 (0, 0), (0, 1), (1, 1), (1, 2), (2, 2) 경로로 나오게 되어 종료가 된다.

<참고> 또한 여기서 오른쪽, 아래로만 이동을 한다는 조건으로 알아두면 좋은 것은 경로의 수는 언제나 (2n-1)(n은 행과 열이 같을 때 그 수)가 나오게 된다는 점도 짐작할 수 있다.

4. 조약돌 두기 문제

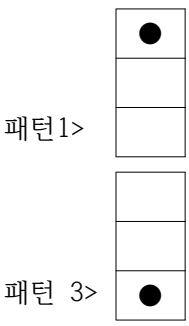
아까 거북이 행렬 문제는 그나마 머리를 쓰면 괜찮은 문제로 생각이 들었다. 하지만 조약돌 두는 문제에 대해서는 큰 어려움이 있을 것이다. 이에 대한 문제를 적어두면서 쉽게 이해를 해보도록 하자.

<조약돌 놓기 문제>
3행 n열 테이블에 각 칸에 정수가 기록이 되어있다. 여기서 정수는 양수, 0, 음수 모두 적용 된다. 이 테이블에 조약돌을 놓는 방법은 각 열엔 적어도 1개 이상의 조약돌을 놓아야 하고, 가로나 세로로 인접한 2칸에 동시에 조약돌을 놓을 수가 없다.

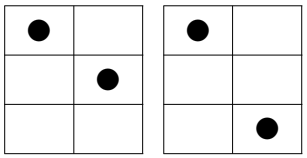
3	10	-1	0
-2	8	4	-1
5	-2	5	-2

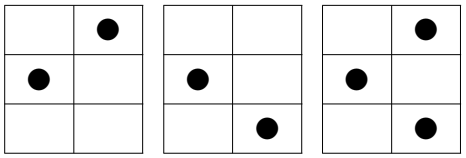
행렬은 요약정리를 위해서 간단하게 3 by 4로 잡았다. 이를 사례로 조약돌이 놓인 자리에 있는 수의 합이 최대가 되는 것을 목표로 한다.

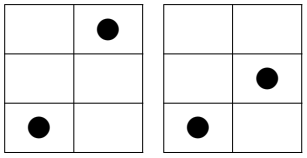
4-1) 조약돌의 패턴을 잡아보자.
임의의 열을 채울 수 있는 패턴은 어떻게 될까? 우선 조약돌을 적어도 1개씩 넣는 것을 조건을 내걸었다면 2^3-1로 7가지 방법이 있다. 하지만, 세로가 인접하게 되면 안 되니 이처럼 4가지 방안을 생각해 볼 수 있겠다.

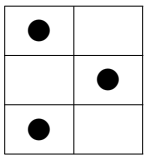


그렇지만 가로에 대해서 인접을 하면 안 된다는 점도 있다. 방금 전에 둘 수 있는 패턴은 4가지로 서로 둘 수 있는 방법이 16가지로 나오게 되는데, 여기서 같이 불어 있는 패턴의 수는 (패턴 x, 패턴 y)로 가정을 하면 (1, 1), (2, 2), (3, 3), (4, 4), (1, 4), (3, 4), (4, 1), (4, 3)으로 이렇게 8가지는 제외를 하고 문제에 임하면 되겠다.


패턴 1에서

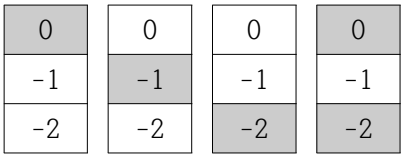

패턴 2에서


패턴 3에서


패턴 4에서

여기서 중요한 것은 현재 패턴과 이전 패턴에 대해서 재귀적으로 비교하는 기능이 있어야 한다는 점을 깨닫고 넘어가라는 뜻이다. 이처럼 놓은 조약돌에 대해서 각 열의 점수를 골라오는 것이 관건이다.

4-2) 각 열 별 점수를 가져오자
y 열에 패턴 1부터 4까지 조약돌이 놓인 경우에 y 열에 놓인 곳의 점수의 합을 구하는 메소드를 만들어야 겠다. 예를 들어 0, -1, -2가 있는 열로 사례를 들어 보자.



이처럼 두면 열_점수(3, 1)은 0, 열_점수(3, 2)은 -1, 열_점수(3, 2)은 -2, 열_점수(3, 3)은 -2가 나오게 되어 최댓값인 0이 나오게 하면 된다.....라고 생각했다면 큰 오산이다. 왜냐면 패턴을 따져야 하기 때문이다. 그래서 맨 앞부분에 인접가능패턴이란 메소드를 따로 만들어서 문제를 해결해 나아갈 필요가 있다.

4-3) 경로 별 점수를 구현해보자.

```
static int 전체_점수(int y, int p) throws Exception{
  if(y==0) return 열_점수(y, p);
  int 이전열_최고점수=Integer.MIN_VALUE;
  for(int k=1;k<=4;k++){
    if(인접가능패턴(p, k)){
      int 점수=전체_점수(y-1, q);
      if(점수>이전열_최고점수) 이전열_최고점수=점수;
    }
  }
  return 이전열_최고점수+열_점수(c, p);              }
```

```
static int 전체_점수(int y) throws Exception{
  int 최고점수=Integer.MIN_VALUE;
  for(int k=1;k<=4;k++){
    int 점수=전체_점수(y, k);
    if(점수>최고점수) 최고점수=점수;
  }
  return 최고점수;
}
```

경로_점수 메소드는 2가지로 나뉠 수 있겠다. 우선 마지막 열부터 조약돌을 패턴 별로 던져보면서 최댓값을 가져오는 경로_점수(int y) 메소드, 이전 패턴들과 비교를 하는데 필요로 한 경로_점수(int y, int p) 메소드. 5페이지에 있는 경로_점수 메소드는 맨 처음부터 시작을 해야겠다. 그러면 어떠한 원리로 돌아가는지에 대해서 지금 요약정리에 있는 행렬을 통해서 알아보도록 하자.

4-4) 조약돌 알고리즘의 돌아가는 원리

(3)	10
-2	8
5	-2

3

3	10
(-2)	8
5	-2

-2

3	10
-2	8
(5)	-2

5

(3)	10
-2	8
(5)	-2

8

우선 0번째 열부터 시작을 하면 패턴 1부터 패턴 4까지 값을 나눠어 보면 3, -2, 5, 8를 골라낼 수 있다. 그렇지만 인접 패턴에 따라서 최댓값을 골라내는 방법이 달라진다.

(3)	10
-2	(8)
5	-2

11

(3)	10
-2	8
5	(-2)

1

3	(10)
(-2)	8
5	-2

8

3	10
(-2)	8
5	(-2)

-4

3	(10)
(-2)	8
5	(-2)

6

3	(10)
-2	8
(5)	-2

15

3	10
-2	8
(5)	(-2)

3

(3)	10
-2	(8)
(5)	-2

16

이처럼 각 인접 패턴 당 최대 점수는 11, 8, 15, 16이 나오게 된다. 이를 이용해서 2번째 열에 대해서 접근을 해보도록 하자. 여기서 짚고 넘어가야 하는 것은 첫 번째 열에서 최댓값이 허용된 패턴의 번호는 1, 2번이 되겠다. 이를 기준으로 경우의 수를 나눠어서 보면 오히려 쉽게 보는데 도움이 되겠다.

-> 위쪽 참고 ->

(10)	-1
8	(4)
-2	5

Max는 19

(10)	-1
8	4
-2	(5)

Max는 20

10	(-1)
(8)	4
-2	5

Max는 15

10	(-1)
(8)	4
-2	(5)

Max는 20

10	-1
(8)	4
-2	(5)

Max는 21

그러면 여기서 최댓값이 허용되는 패턴은 3이다. 이를 통해서 마지막 열에 있는 값들을 통해서 결과 값을 출력해보자.

-1	(0)
4	-1
(5)	-2

Max는 21

-1	0
4	(-1)
(5)	-2

Max는 20

이렇게 한다면 최종 결과는 21이 나오게 된다. 이처럼 최댓값을 먼저 따지고, 거기에 있는 패턴을 통해서 구분을 하면 알고리즘이 돌아가는 원리에 대해서 익히는데 도움이 될 것이다. 그럼 최종적으로 돌린 결과는 아래와 같이 나온다.

(3)	10	-1	(0)
-2	(8)	4	-1
(5)	-2	(5)	-2

4-5) 인접가능패턴 메소드와 열_점수 메소드 정리하기

```
static boolean 인접가능패턴(int p1, int p2){
  if(p1==p2) return false;
  int[][] patterns={{1, 4}, {3, 4}, {4, 1}, {4, 3}};
  for(int k=0;k<patterns.length;k++){
    if(patterns[k][0]==p1 && patterns[k][1]==p2)
      return false;
  }
  return true;
}
```

```
static int 열_점수(int y, int p) throws Exception{
  switch(p) {
    case 1 : return arr[0][y];
    case 2 : return arr[1][y];
    case 3 : return arr[2][y];
    case 4 : return arr[0][y]+arr[2][y];
  } throw new Exception("invaild p : "+p);
}
```

4-3 메소드와 4-5 메소드를 합치면 강의 노트에 나와있는 코딩과 유사하게 나오니 자세한 건 강의 노트를 참고하는 것이 도움 되겠다. 여기서 중요한 것은 인접가능패턴 메소드를 통해서 서로 패턴 별로 인접하지 않게 사전에 방지를 하는 것을 목표로 두고 있으며,

열_점수는 각 열 당 점수를 골라내서 출력을 하는 것을 목표로 두고 있다. 그렇지만 4-3, 4-5 메소드들을 합쳐서 쓰게 되면 재귀 중복이 많이 일어나게 된다. 그래서 동적 프로그래밍으로 재귀 중복을 줄여보는 연습이 필요로 하다.

4-6) 동적 프로그래밍으로 바꿔서 풀어보자

```
public class Pebble_Test {

    static int[][] a = {
            {  6,  7, 12, -5, 5,  3, 11, 3 },
            { -8, 10, 14,  9, 7, 13,  8, 5 },
            { 11, 12,  7,  4, 8, -2,  9, 4 }};

    static int[][] 경로점수 = new int[8][5];

    static boolean 인접가능패턴(int 패턴1, int 패턴2) {
        if (패턴1 == 패턴2) return false; // 동일한 패턴은 인접할 수 없다.
        int[][] patterns = { {1, 4}, {3, 4}, {4, 1}, {4, 3} }; // 인접할 수 없는 패턴
        for (int i = 0; i < patterns.length; ++i)
            if (패턴1 == patterns[i][0] && 패턴2 == patterns[i][1])
                return false;
        return true;
    }

    static int 열_점수(int c, int p) throws Exception {
        switch (p) {
        case 1: return a[0][c]; // 패턴1
        case 2: return a[1][c]; // 패턴2
        case 3: return a[2][c]; // 패턴3
        case 4: return a[0][c] + a[2][c]; // 패턴4
        }
        throw new Exception("invalid p:" + p);
    }

    static int 호출횟수 = 0;

    static int 전체_점수(int c, int p) throws Exception {
        if (경로점수[c][p] != 0) return 경로점수[c][p];

        ++호출횟수;
        if (c == 0) return 경로점수[c][p] = 열_점수(c, p);
        int 이전열_최고점수 = Integer.MIN_VALUE;
        for (int q = 1; q <= 4; ++q) // q는 c-1 열의 패턴
            if (인접가능패턴(p, q)) {
                int 점수 = 전체_점수(c - 1, q);
                if (점수 > 이전열_최고점수) 이전열_최고점수 = 점수;
            }
        return 경로점수[c][p] = (이전열_최고점수 + 열_점수(c, p));
    }

    static int 전체_점수(int c) throws Exception {
        int 최고점수 = Integer.MIN_VALUE;
        for (int p = 1; p <= 4; ++p) {
            int 점수 = 전체_점수(c, p);
            if (점수 > 최고점수) 최고점수 = 점수;
        }
        return 최고점수;
    }

    public static void main(String[] args) throws Exception {
        System.out.printf("경로점수=%d\n", 전체_점수(7));
        System.out.printf("호출횟수=%d\n", 호출횟수);
    }
}
```

여기서 중요한 것은 경로 점수에 대해서 패턴 별로 따로 저장을 하고 난 후에 출력을 하는 것이다. 이처럼 출력을 하게 되면 아까와 마찬가지고 공간 복잡도는 늘어나게 되지만, 시간 복잡도가 줄어드는 목적으로 구현을 할 수 있는 방법이 바로 이 방법이다. 전체 점수를 불러오는 방법에 대해서는 패턴 별로 허용된 것들의 최댓값만 저장을 하는 것이 목표이기 때문에 최댓값 골라내는 알고리즘을 연상하면 쉽게 생각할 수 있을 것이다. 그러면 아까 재귀 호출을 해서 여러 방법으로 풀이하는 수가 많아지는데 이를 호출하게 되면 오히려 재귀 호출의 수가 줄어드는데 큰 도움을 준다.

<참고> Integer.MIN_VALUE, Integer.MAX_VALUE 은 무엇인가요??
우리가 int 형 변수를 쓰다 보면 4바이트의 크기로 구성된 정수형 변수인 것을 짐작할 수 있을 것이다. 여기서 이용한 MIN_VALUE는 -2147483648로 -2^{31}이 되고, MAX_VALUE는 2147483647로서 $2^{31}-1$이 되겠다. 여기서 MIN_VALUE를 쓴 이유가 행렬에 int 형보다 더 작은 숫자가 들어갈 수가 없고, 또한 이번 행렬에는 음수도 포함되어 있어서 최댓값을 골라내는데 있어서 이상 없이 골라내기 위해 쓴 이유로 생각을 해보면 되겠다. 또한 MAX_VALUE는 이진 탐색 트리에서 Dummy 개념을 봤으면 흔히 봤을 것이다. 이처럼 알고리즘에서도 int에서 가장 큰 값, 작은 값을 통해서 알고리즘이 올바르게 돌아가는데 있어서 변색 없이 이용할 수 있다는 점을 숙지하고 넘어가면 좋겠다.